Theory and Frontiers of Computer Science

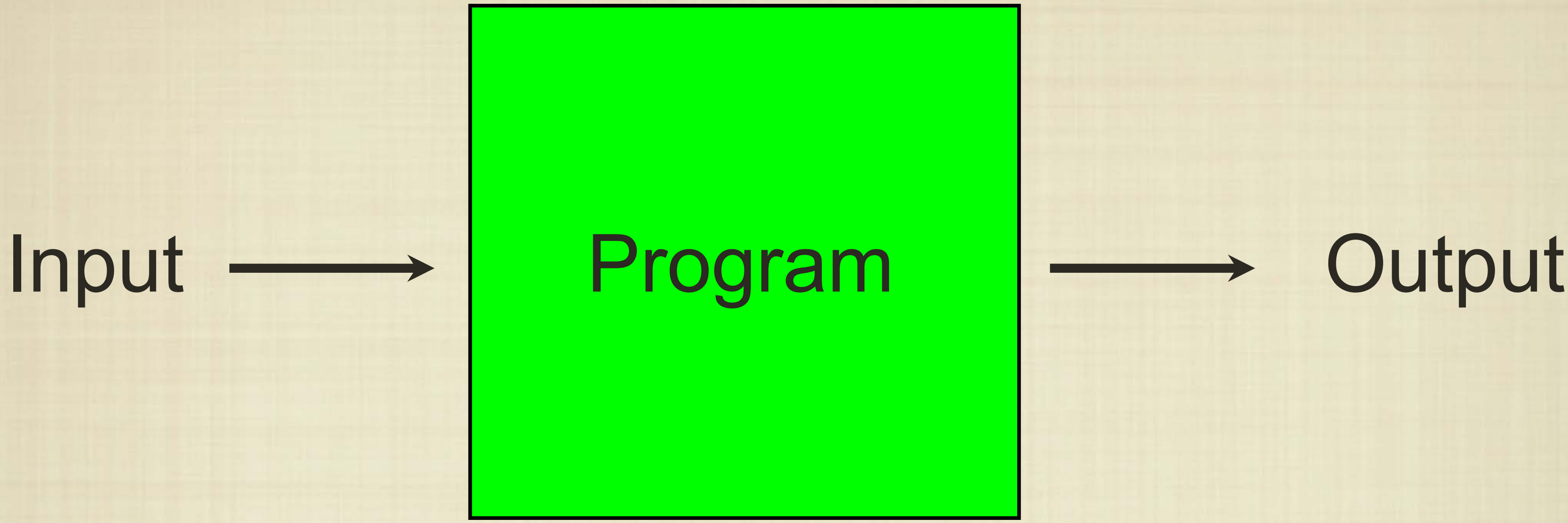
Fall 2013

Carola Wenk

We have seen so far…

- Computer Architecture and Digital Logic (Von Neumann Architecture, binary numbers, circuits)
- Introduction to Python (if, loops, functions)
- Algorithm Analysis (Min, Searching, Sorting; Runtimes)
- Linked Structures (Lists, Trees, Huffman Coding)
- Graphs (Adjacency Lists, BFS, Connected Components)
- Data Mining (Finding patterns, supervised learning)

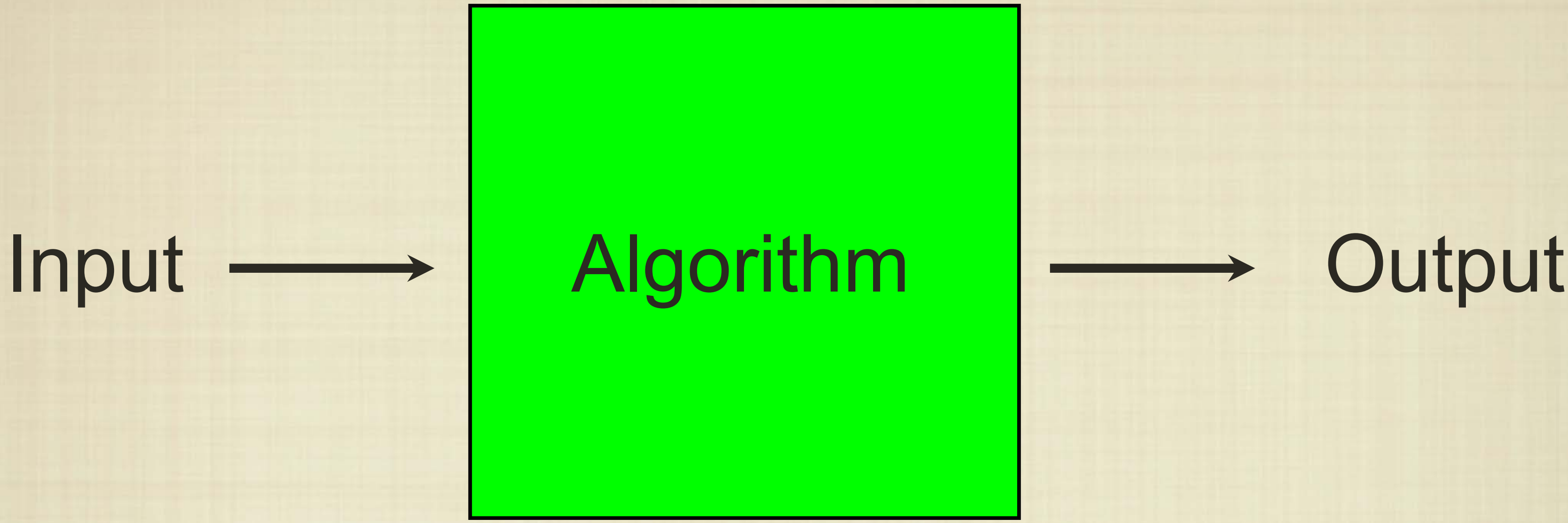
The Big Picture



So far, we have been designing algorithms for problems that meet given specifications.

There are many programs that can implement a particular algorithm, but we can make our picture even more abstract.

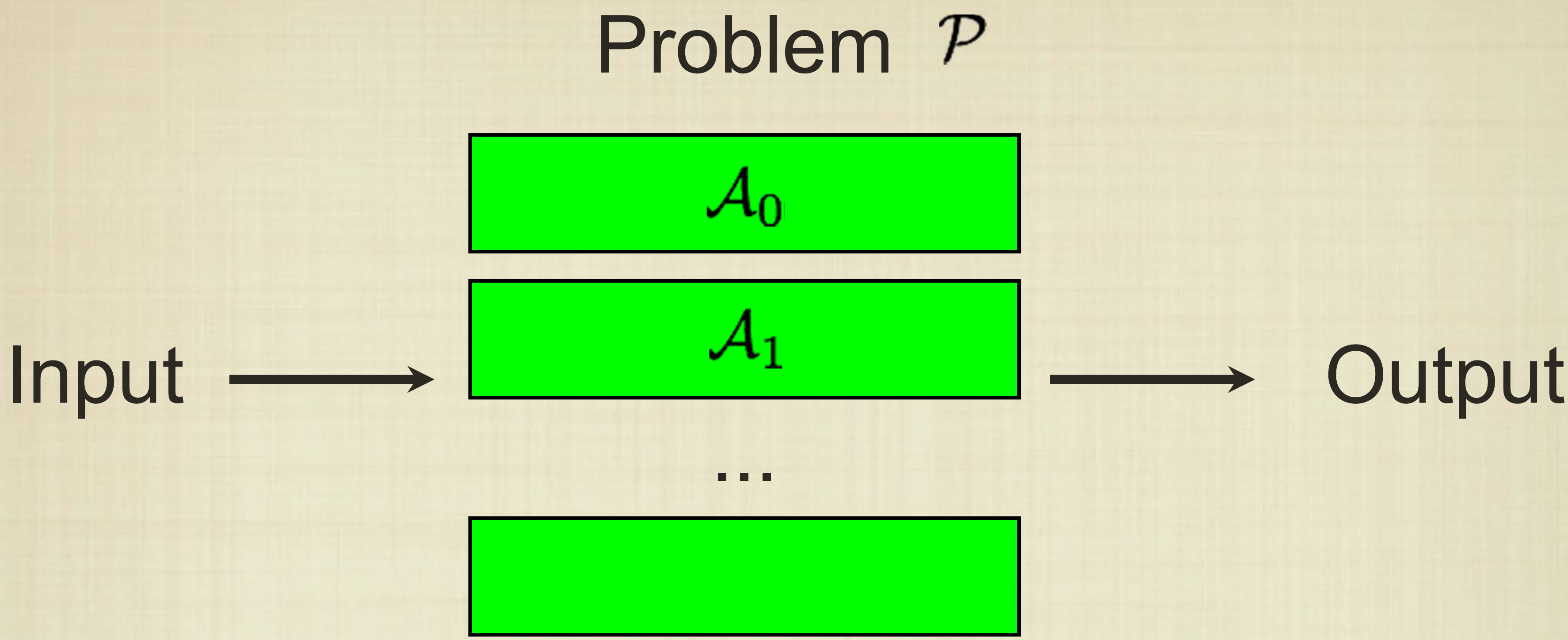
The Big Picture



We can think even more abstractly: for any particular problem we can come up with many algorithms.

A natural way to categorize algorithms is by the problems they solve.

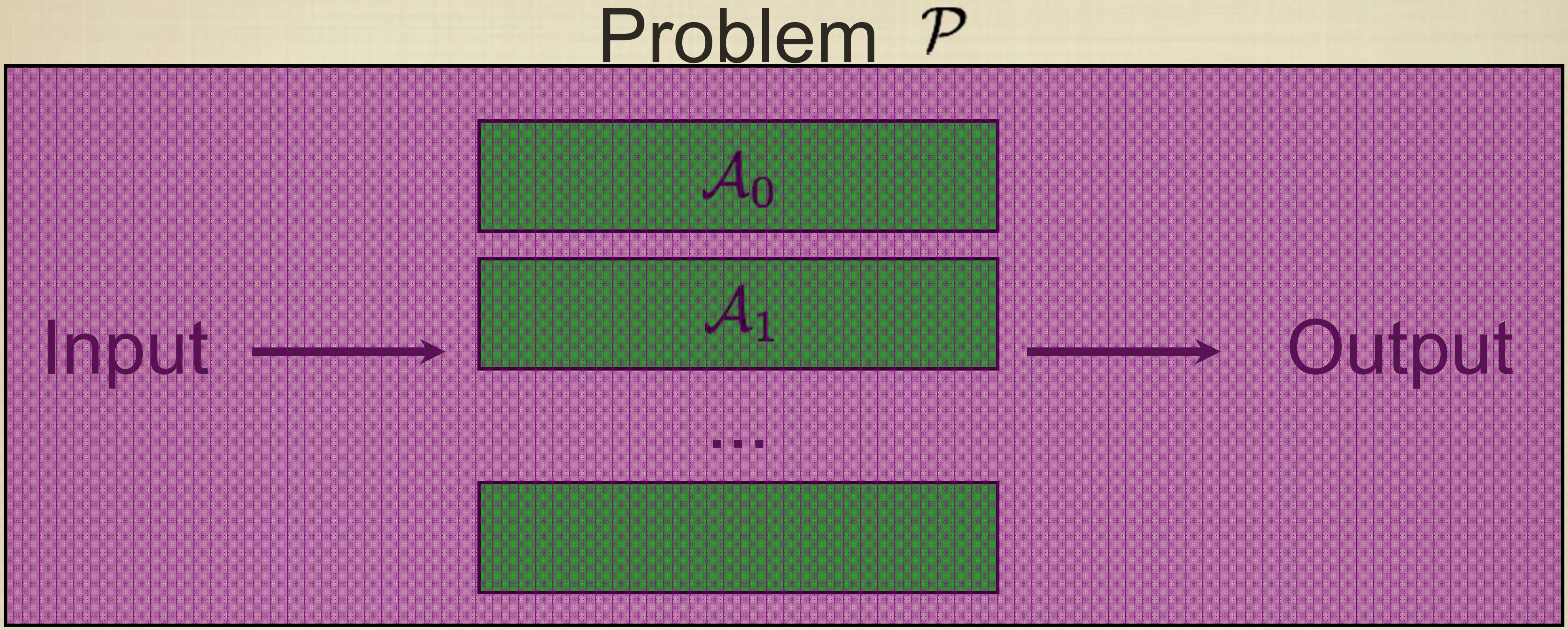
The Big Picture



We can think even more abstractly: for any particular problem we can come up with many algorithms.

A natural way to categorize algorithms is by the problems they solve.

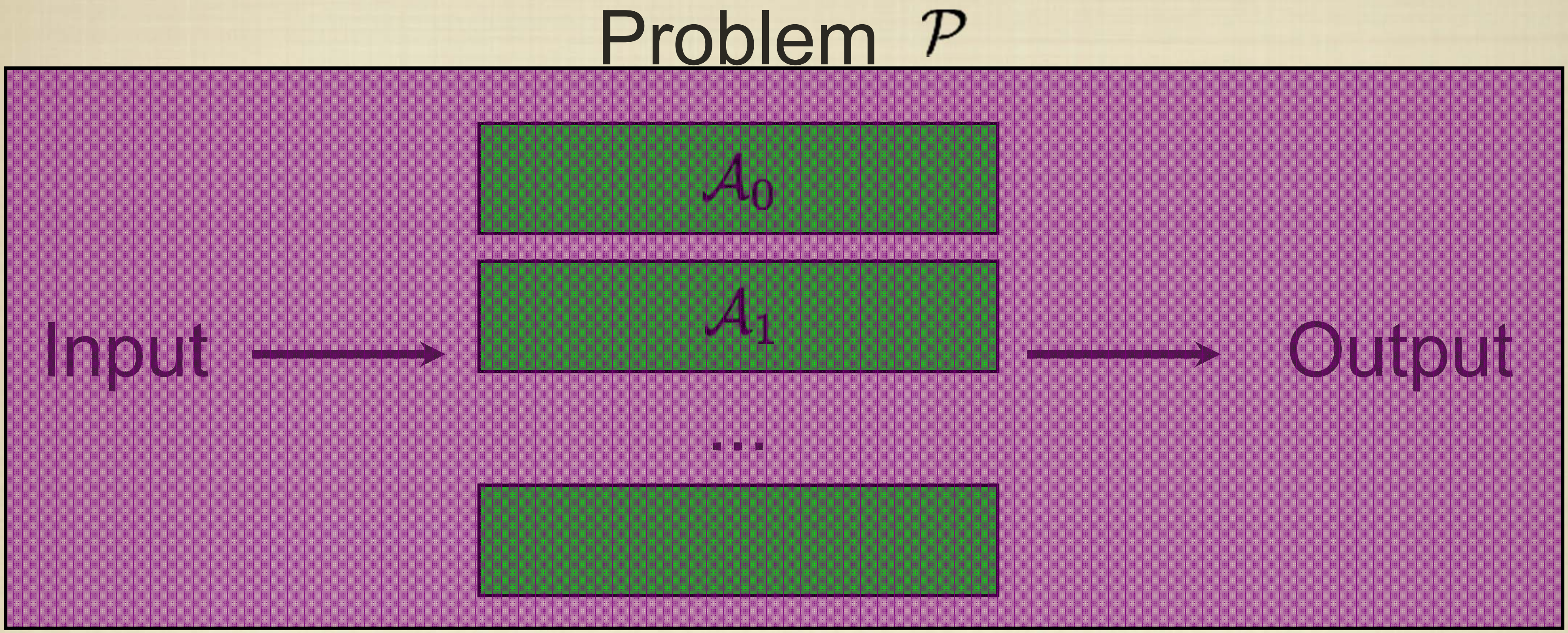
The Big Picture



We can think even more abstractly: for any particular problem we can come up with many algorithms.

A natural way to categorize algorithms is by the problems they solve.

The Big Picture



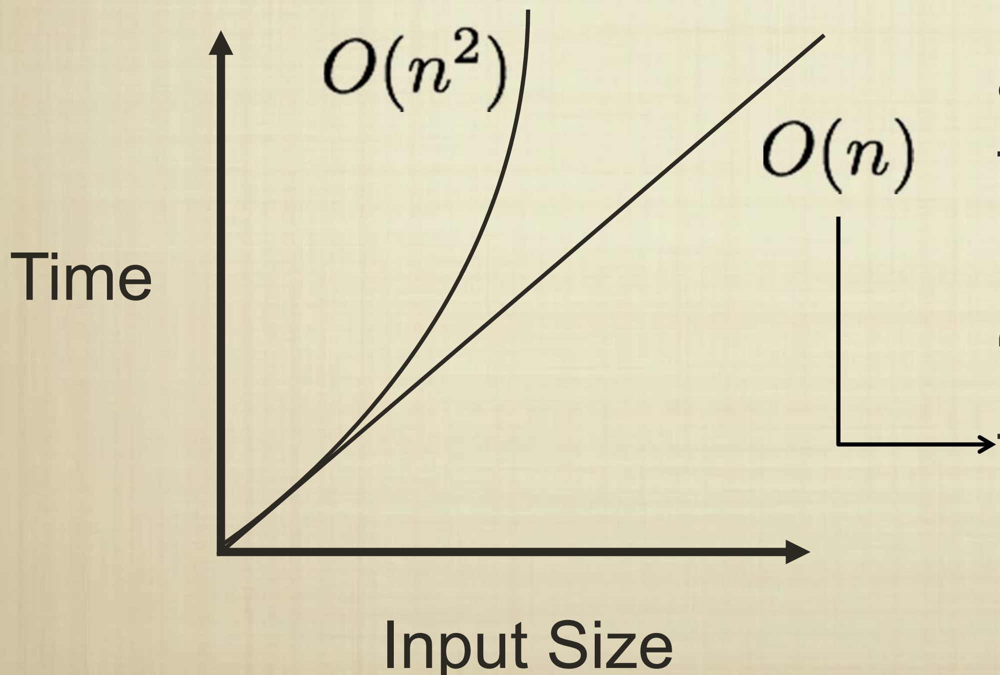
Then, for a particular problem $\mathcal{P}$, we are interested in finding an "efficient" algorithm.

Is this always possible? What does "efficient" mean?

(Worst-Case) Asymptotic Runtime Analysis

Usually, the abstract performance of an algorithm depends on the actual input for any particular size n.

Which inputs should we use to characterize runtime?

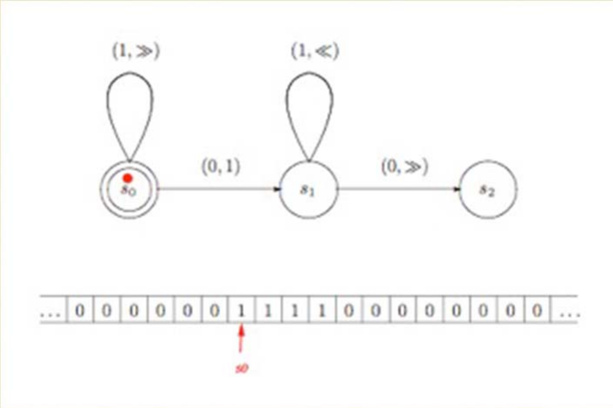


We define algorithm performance as conservatively as possible, on the worst-case inputs.

"No matter what, my algorithm takes at most c·*n* steps for an input size of *n*."

Computational Complexity

The field of “computational complexity” tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.



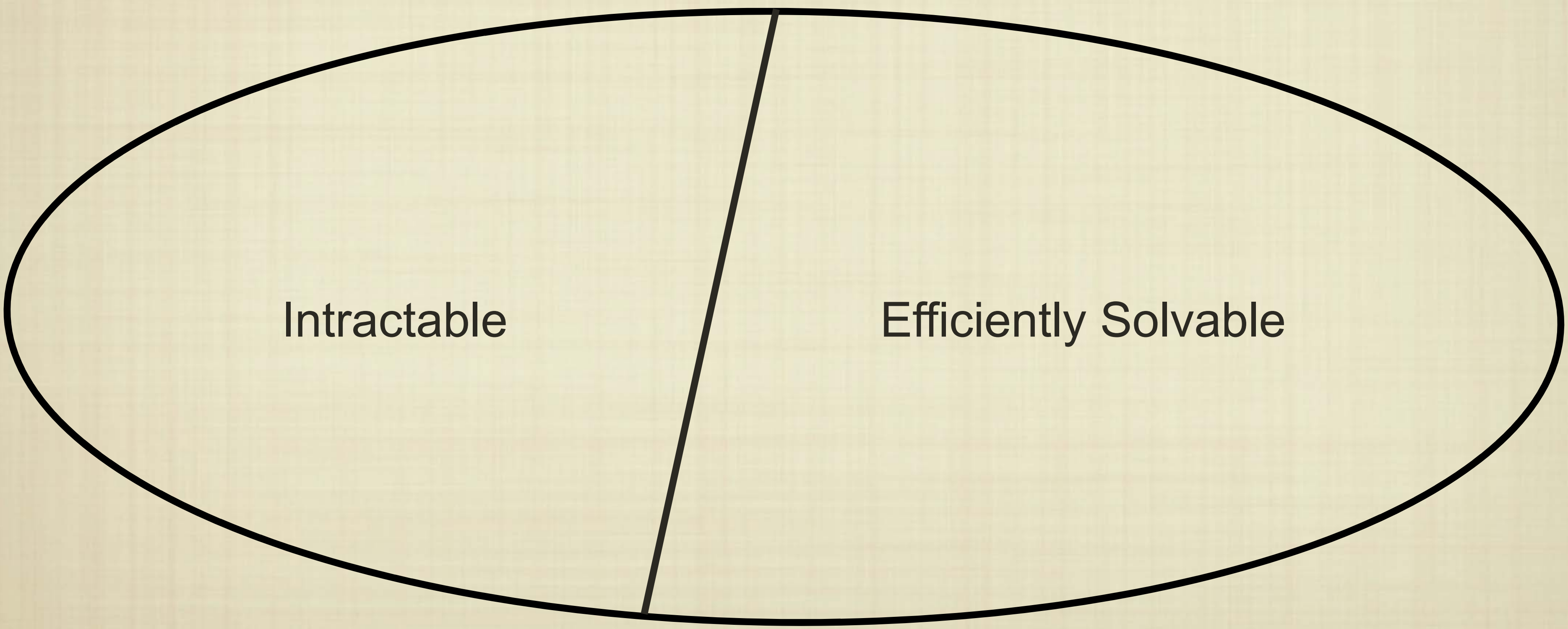
Alan Turing

A **Turing Machine** captures the essential components of computation: memory and state information.

The Church-Turing Thesis states that “everything algorithmically computable is computable by a Turing machine.”

Computational Complexity

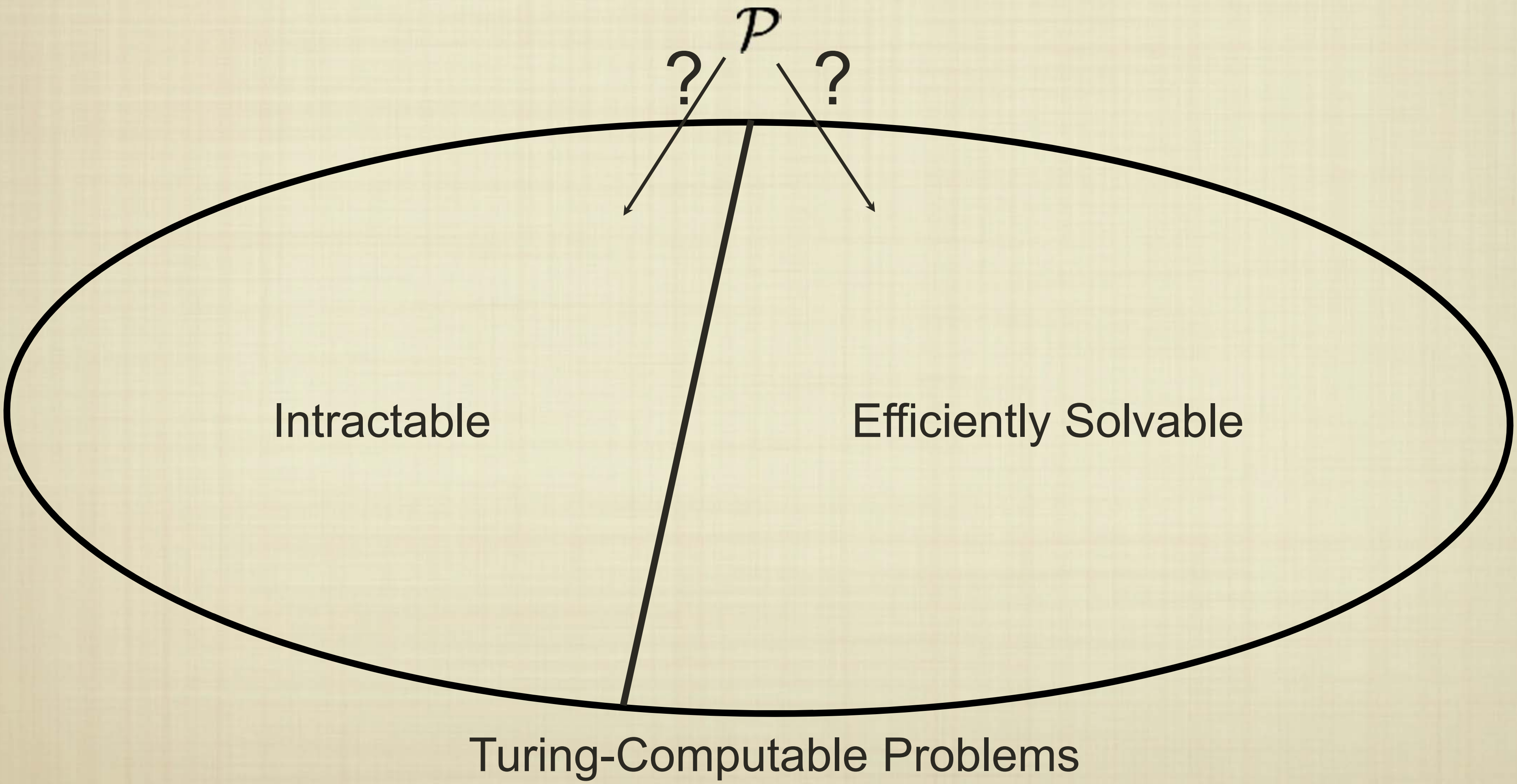
The field of “computational complexity” tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.



Turing-Computable Problems

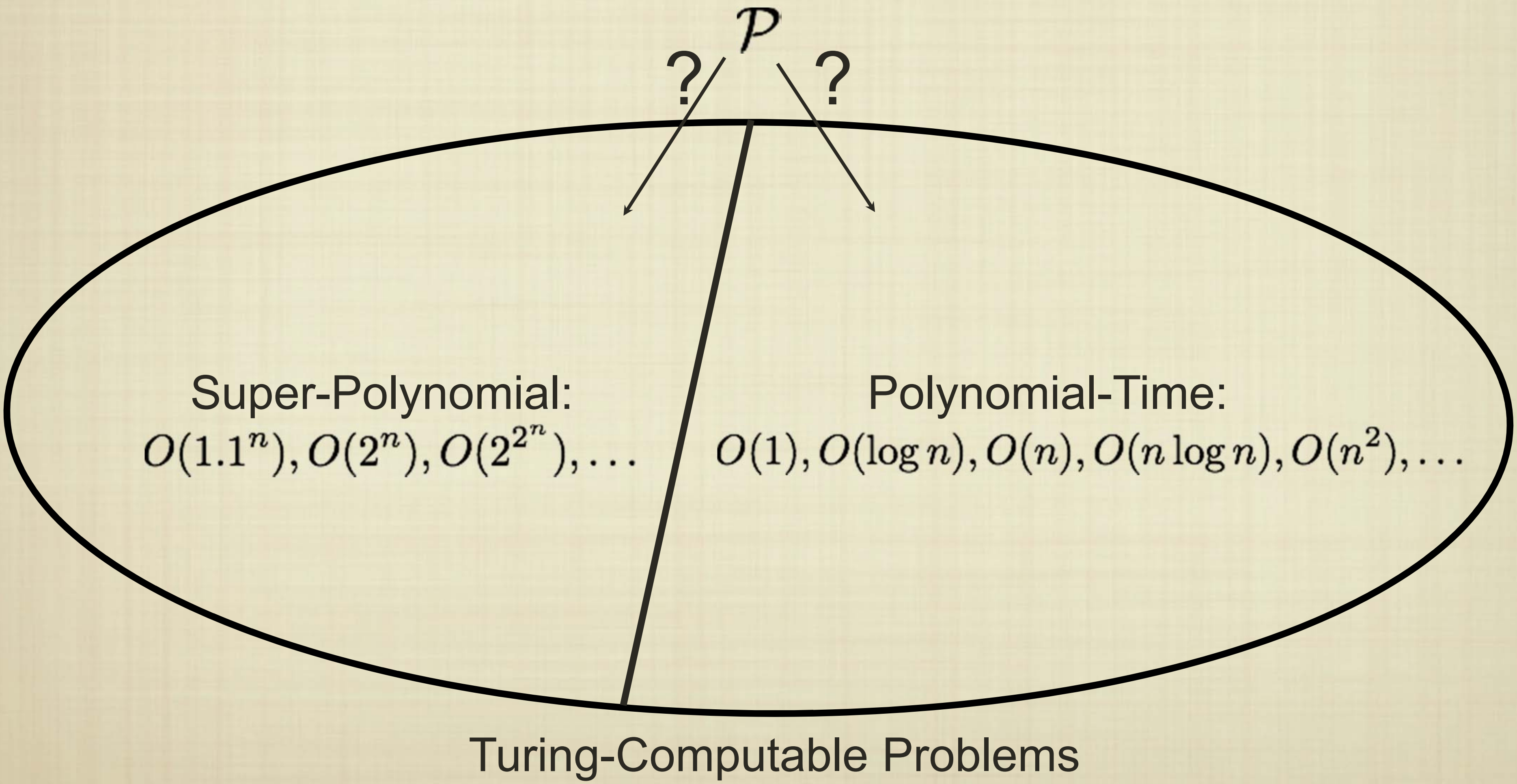
Computational Complexity

The field of “computational complexity” tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.

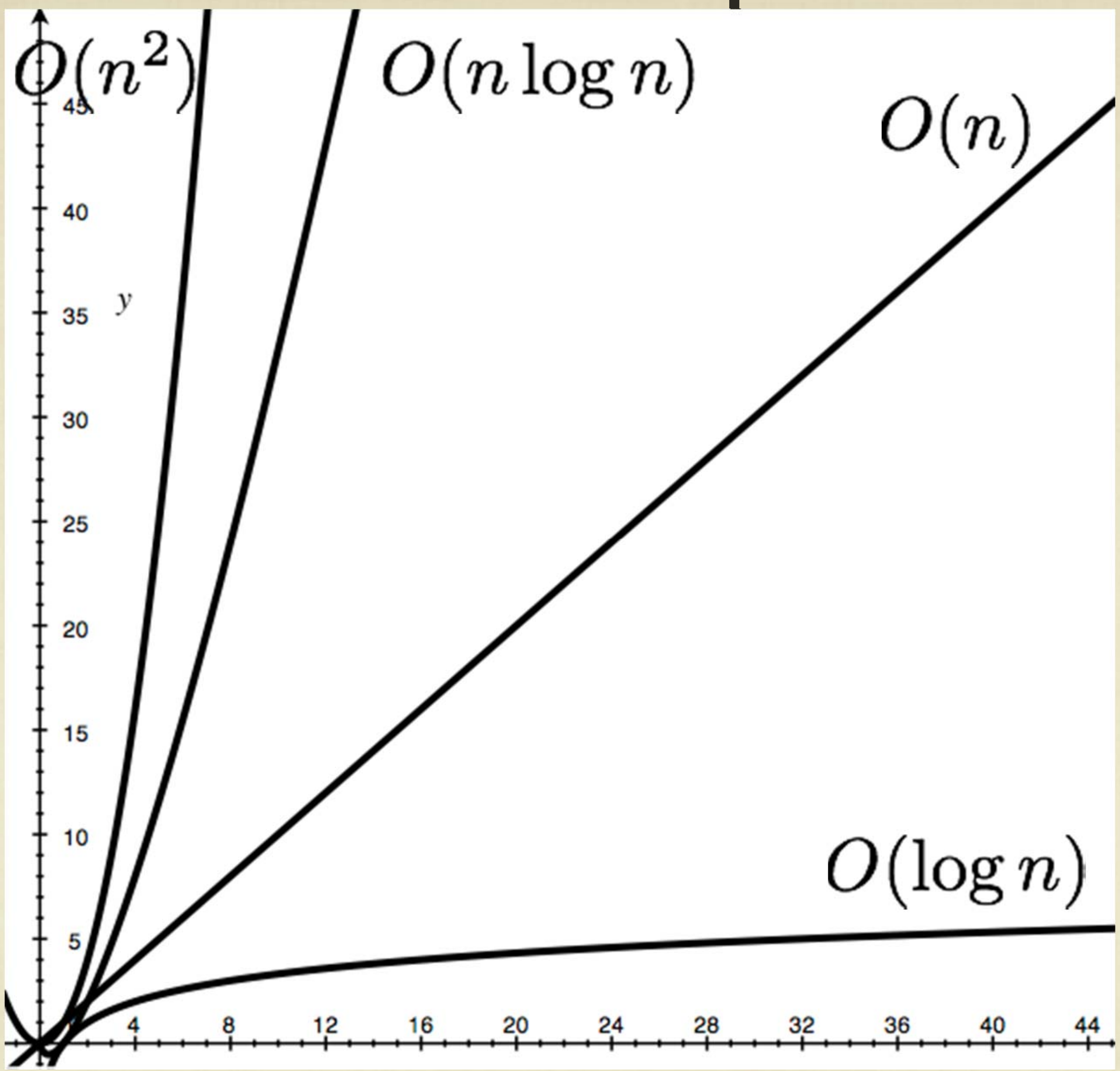


Computational Complexity

The field of “computational complexity” tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.

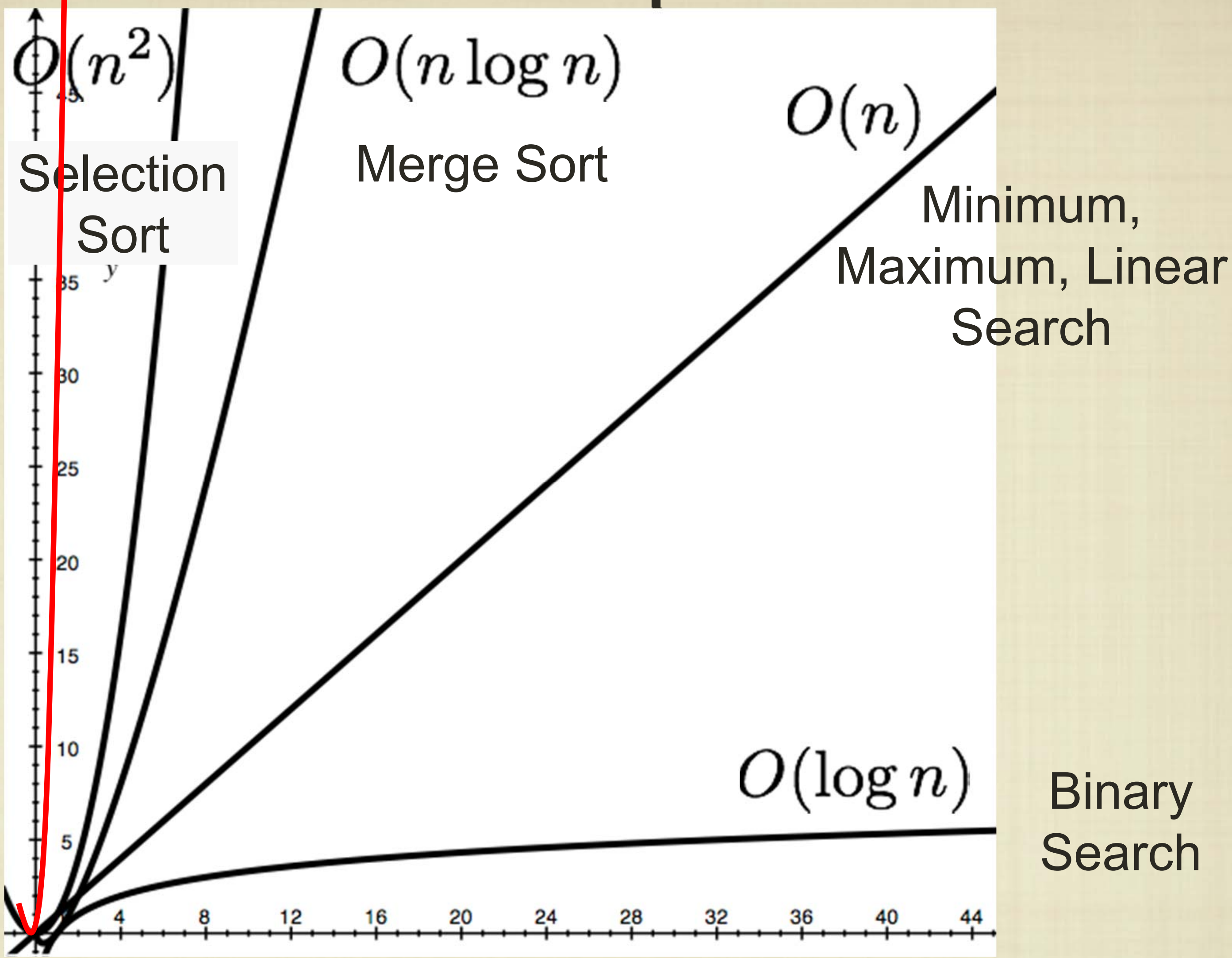


Polynomial Versus Exponential Time



We adopt the convention that as long as an algorithm's running time is polynomial (or logarithmic) in the input, it is "efficient". Why is this a good criterion?

Polynomial Versus Exponential Time



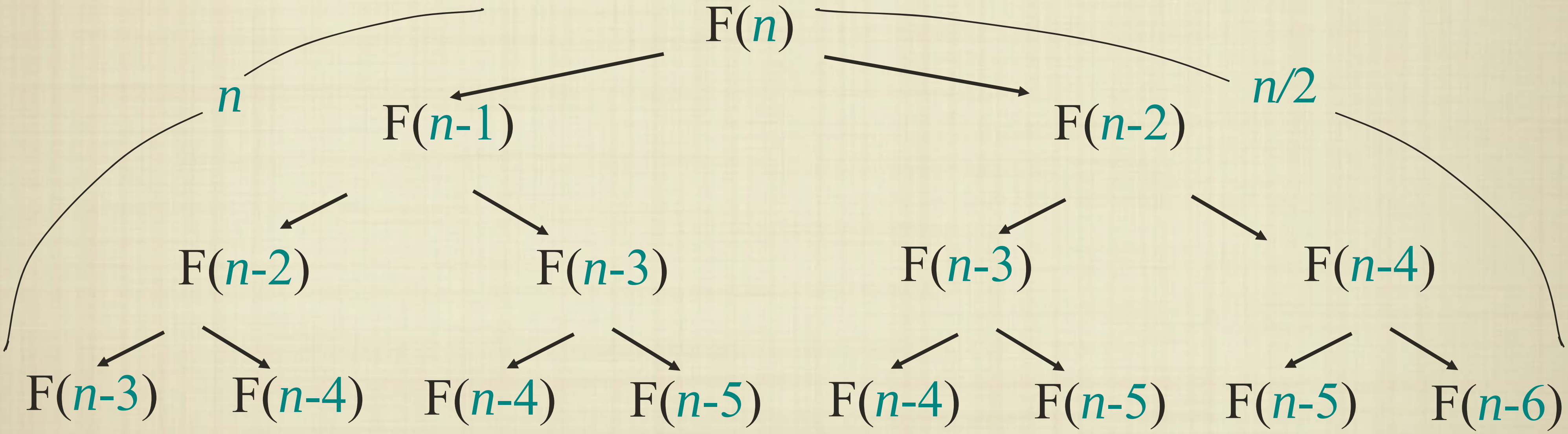
We adopt the convention that as long as an algorithm's running time is polynomial (or logarithmic) in the input, it is "efficient". Why is this a good criterion?

Example: Fibonacci numbers

$F(0)=0$; $F(1)=1$; $F(n)=F(n-1)+F(n-2)$ for $n \geq 2$

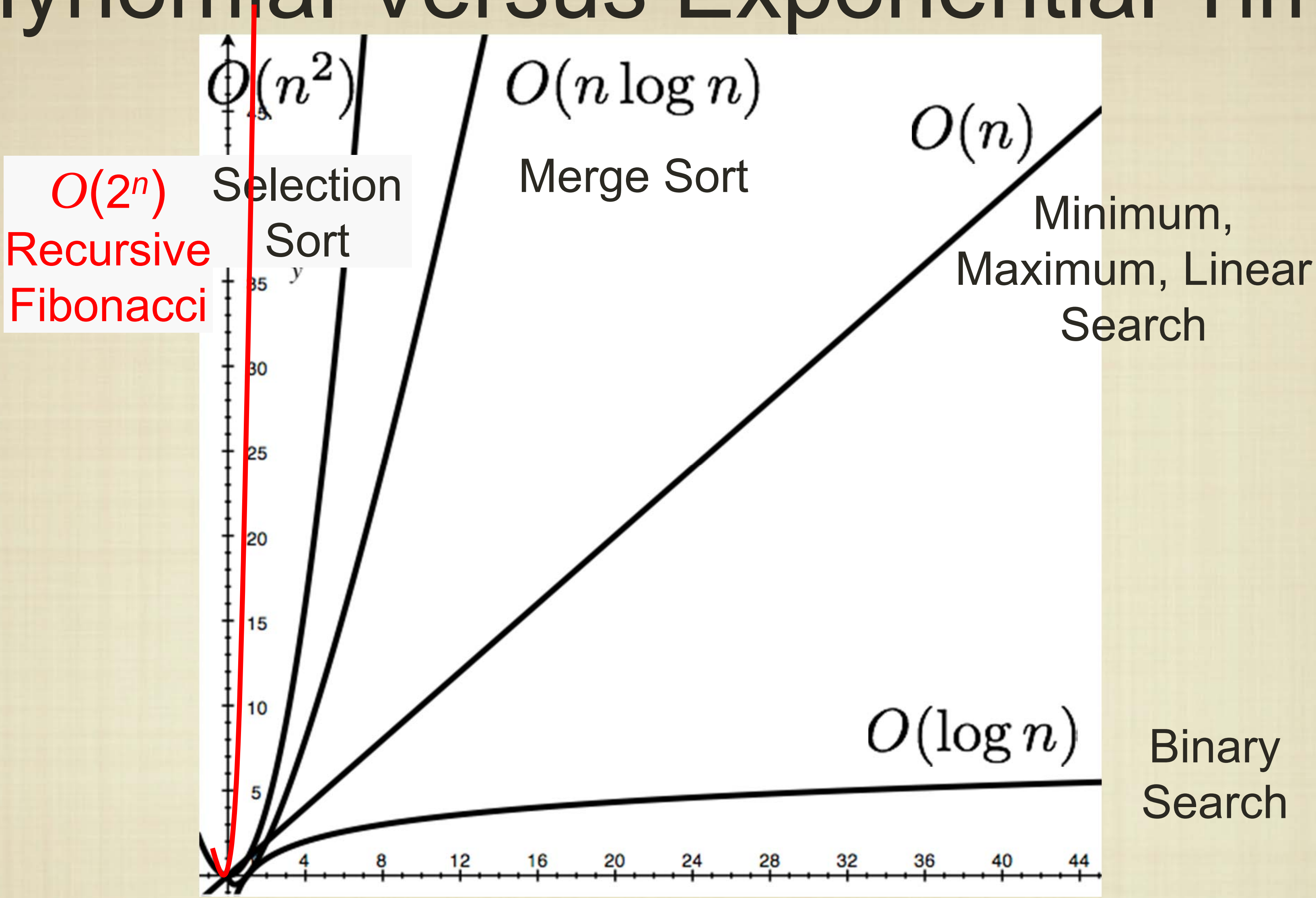
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Implement this recursion directly:



Runtime is exponential: $2^{n/2} \leq T(n) \leq 2^n$

Polynomial Versus Exponential Time



We adopt the convention that as long as an algorithm's running time is polynomial (or logarithmic) in the input, it is "efficient". Why is this a good criterion?

Polynomial versus Exponential Time

Suppose we have two algorithms $\mathcal{A}$ and $\mathcal{B}$ for the same problem, where:

$$T_{\mathcal{A}}(n) = n^{1000}$$

$$T_{\mathcal{B}}(n) = 2^{0.000001n}$$

Which algorithm is better according to our usual method of comparison? For all *large* n?

Polynomial versus Exponential Time

Suppose we have two algorithms $\mathcal{A}$ and $\mathcal{B}$ for the same problem, where:

$$T_{\mathcal{A}}(n) = n^{1000}$$

$$T_{\mathcal{B}}(n) = 2^{0.000001n}$$

Which algorithm is better according to our usual method of comparison? For all *large* n?

$$n^{1000} \overset{\leq}{\underset{\geq}{?}} 2^{0.000001n}$$

Polynomial versus Exponential Time

Suppose we have two algorithms $\mathcal{A}$ and $\mathcal{B}$ for the same problem, where:

$$T_{\mathcal{A}}(n) = n^{1000}$$

$$T_{\mathcal{B}}(n) = 2^{0.000001n}$$

Which algorithm is better according to our usual method of comparison? For all *large* n?

$$n^{1000} \underset{\geq}{\overset{\leq}{?}} 2^{0.000001n}$$

$$1000 \cdot \log_2 n \underset{\geq}{\overset{\leq}{?}} 0.000001n$$

Polynomial versus Exponential Time

Suppose we have two algorithms $\mathcal{A}$ and $\mathcal{B}$ for the same problem, where:

$$T_{\mathcal{A}}(n) = n^{1000}$$

$$T_{\mathcal{B}}(n) = 2^{0.000001n}$$

Which algorithm is better according to our usual method of comparison? For all *large* n?

$$n^{1000} \overset{\leq}{\underset{\geq}{?}} 2^{0.000001n}$$

$$10^9 \cdot \log_2 n \overset{\leq}{\underset{\geq}{?}} n$$

Polynomial versus Exponential Time

Suppose we have two algorithms $\mathcal{A}$ and $\mathcal{B}$ for the same problem, where:

$$T_{\mathcal{A}}(n) = n^{1000}$$

$$T_{\mathcal{B}}(n) = 2^{0.000001n}$$

Which algorithm is better according to our usual method of comparison? For all *large* n?

$$n^{1000} \underset{\geq}{\overset{\leq}{?}} 2^{0.000001n}$$

$$10^9 \underset{\geq}{\overset{\leq}{?}} \frac{n}{\log_2 n}$$

Polynomial versus Exponential Time

Suppose we have two algorithms $\mathcal{A}$ and $\mathcal{B}$ for the same problem, where:

$$T_{\mathcal{A}}(n) = n^{1000}$$

$$T_{\mathcal{B}}(n) = 2^{0.000001n}$$

Which algorithm is better according to our usual method of comparison? For all *large* n?

$$n^{1000} \;\overset{\leq}{\underset{\geq}{?}}\; 2^{0.000001n}$$

$$10^9 \leq \frac{n}{\log_2 n}$$

For all *large* n,
e.g., for all $n \geq 10^{11}$

Polynomial versus Exponential Time

Suppose we have two algorithms $\mathcal{A}$ and $\mathcal{B}$ for the same problem, where:

$$T_{\mathcal{A}}(n) = n^{1000}$$

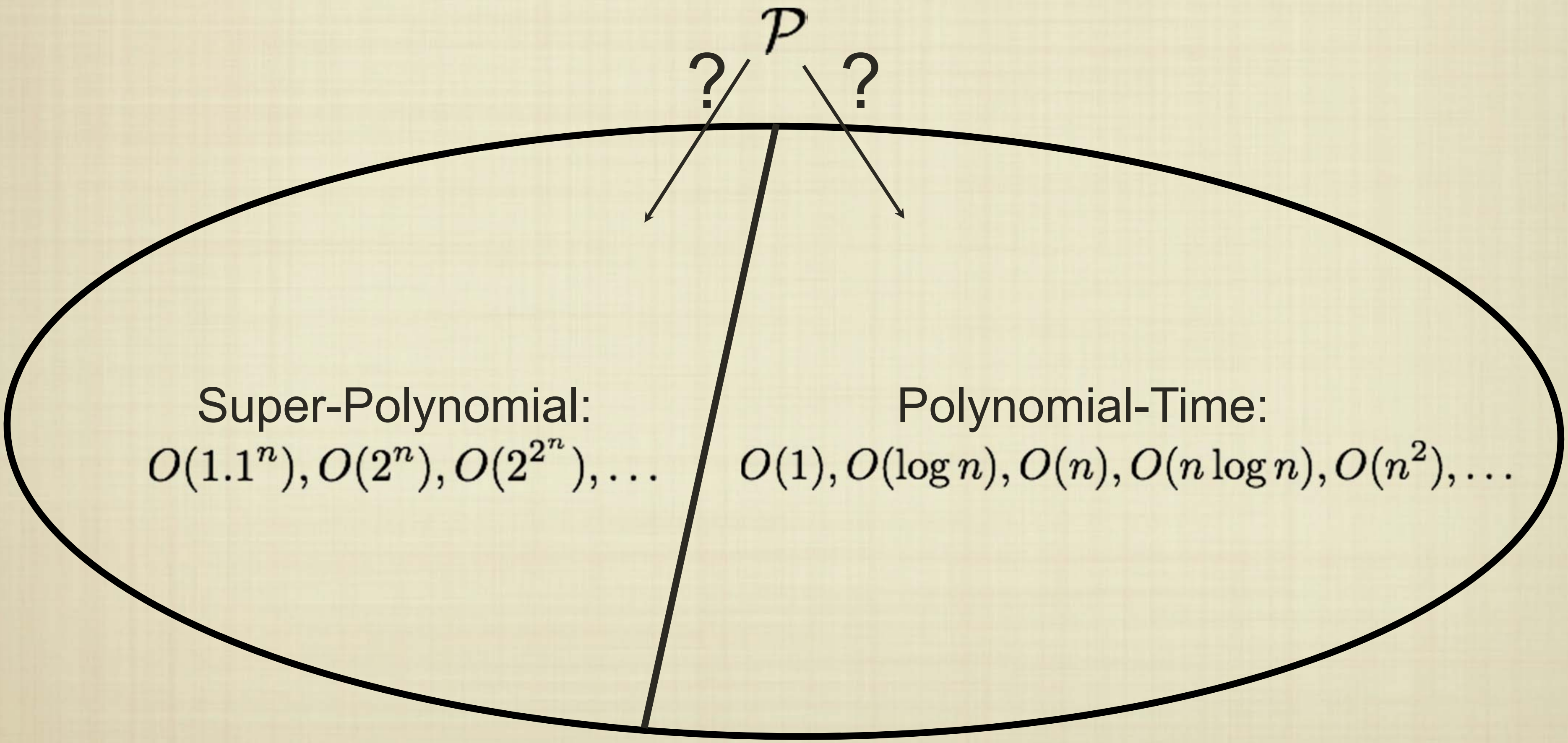
$$T_{\mathcal{B}}(n) = 2^{0.000001n}$$

Actually, every polynomial is (eventually) upper bounded by any exponential.

Lemma: For any $c > 1, x > 0$, and any $\epsilon > 0$, we have that $n^x \leq c^{\epsilon \cdot n}$, for sufficiently large n .

Computational Complexity

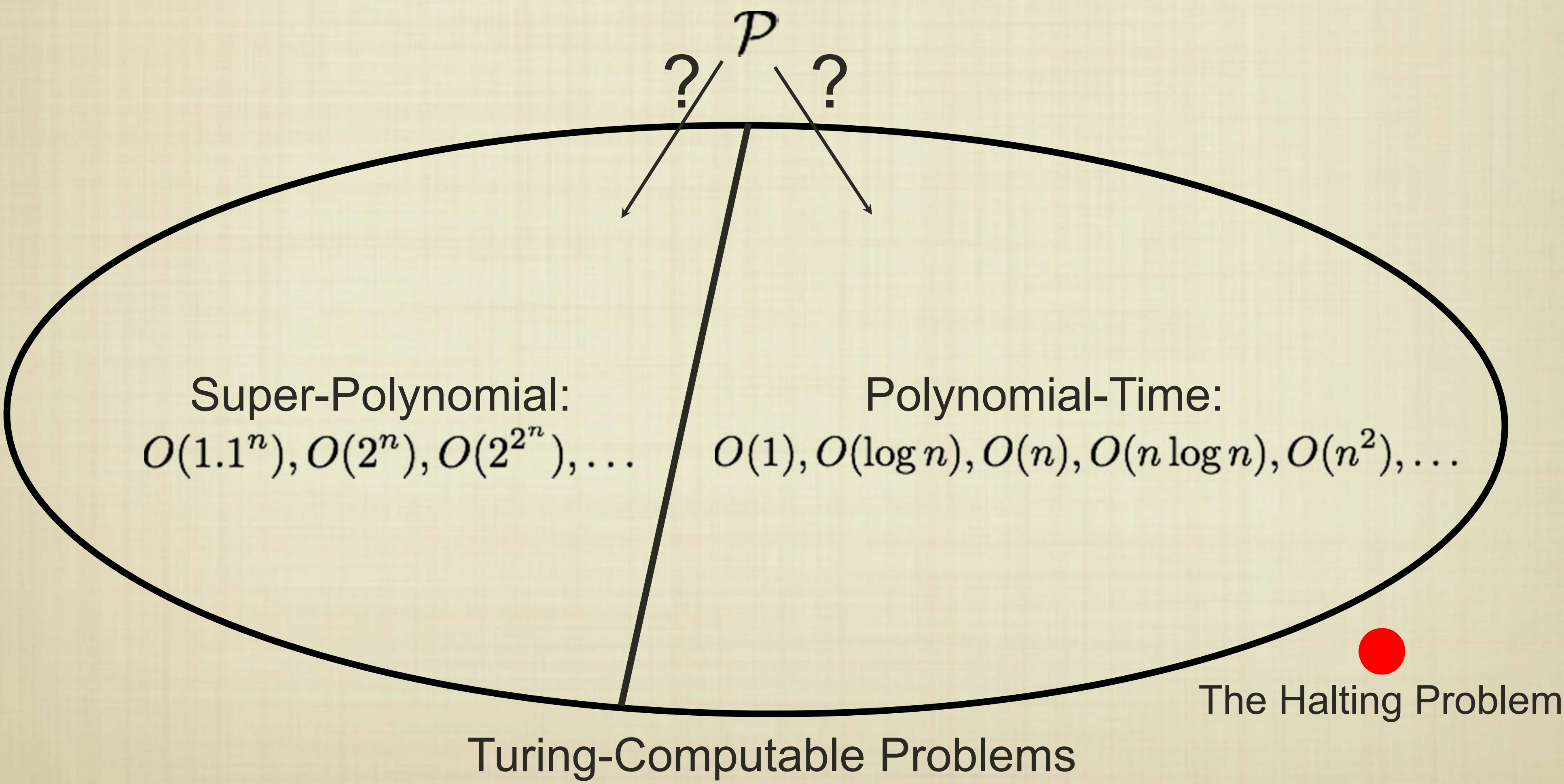
The field of "computational complexity" tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.



Turing-Computable Problems

Computational Complexity

The field of “computational complexity” tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.



Upper and Lower Bounds

If we can come up with an algorithm that correctly solves a particular problem $\mathcal{P}$, then its worst-case running time is an upper bound.

What would be more useful though, is evidence that $\mathcal{P}$ cannot be solved in a given amount of time. In other words, to establish difficulty we need a lower bound on the running time of any algorithm for $\mathcal{P}$.

Upper Bound

Algorithm A for $\mathcal{P}$

↓

$\mathcal{P}$ can be solved in $T_A(n)$ time

Lower Bound

Regardless of the algorithm, the problem $\mathcal{P}$ cannot be solved in less than $T^*(n)$ time.

Upper and Lower Bounds

If we can come up with an algorithm that correctly solves a particular problem $\mathcal{P}$, then its worst-case running time is an upper bound.

What would be more useful though, is evidence that $\mathcal{P}$ cannot be solved in a given amount of time. In other words, to establish difficulty we need a lower bound on the running time of any algorithm for $\mathcal{P}$.

Upper Bound

MergeSort for sorting a list

↓

Sorting can be done in $O(n \log n)$ time

Lower Bound

Every sorting algorithm requires at least ??? time.

Upper and Lower Bounds

If we can come up with an algorithm that correctly solves a particular problem $\mathcal{P}$, then its worst-case running time is an upper bound.

What would be more useful though, is evidence that $\mathcal{P}$ cannot be solved in a given amount of time. In other words, to establish difficulty we need a lower bound on the running time of any algorithm for $\mathcal{P}$.

Upper Bound

MergeSort for sorting a list

↓

Sorting can be done in $O(n \log n)$ time

Lower Bound

Every sorting algorithm requires at least cn time.

Can we match the lower bound to the upper bound?

Lower Bound for Sorting

We came up with an algorithm for sorting that took $O(n \log n)$ time, can we be sure that this is the fastest possible?

Given a list of distinct elements, consider what any algorithm for sorting actually does:

Unsorted List L How many possible orderings?

$\downarrow$

$\mathcal{A}$

$\downarrow$

Sorted List L How many possible outputs?

Lower Bound for Sorting

We came up with an algorithm for sorting that took $O(n \log n)$ time, can we be sure that this is the fastest possible?

Given a list of distinct elements, consider what any algorithm for sorting actually does:

Unsorted List L How many possible orderings?

$n!$

$\downarrow$

$\mathcal{A}$

$\downarrow$

Sorted List L How many possible outputs?

1

Lower Bound for Sorting

Unsorted List L How many possible orderings?

$n!$

↓

$\mathcal{A}$

↓

Sorted List L How many possible outputs?

1

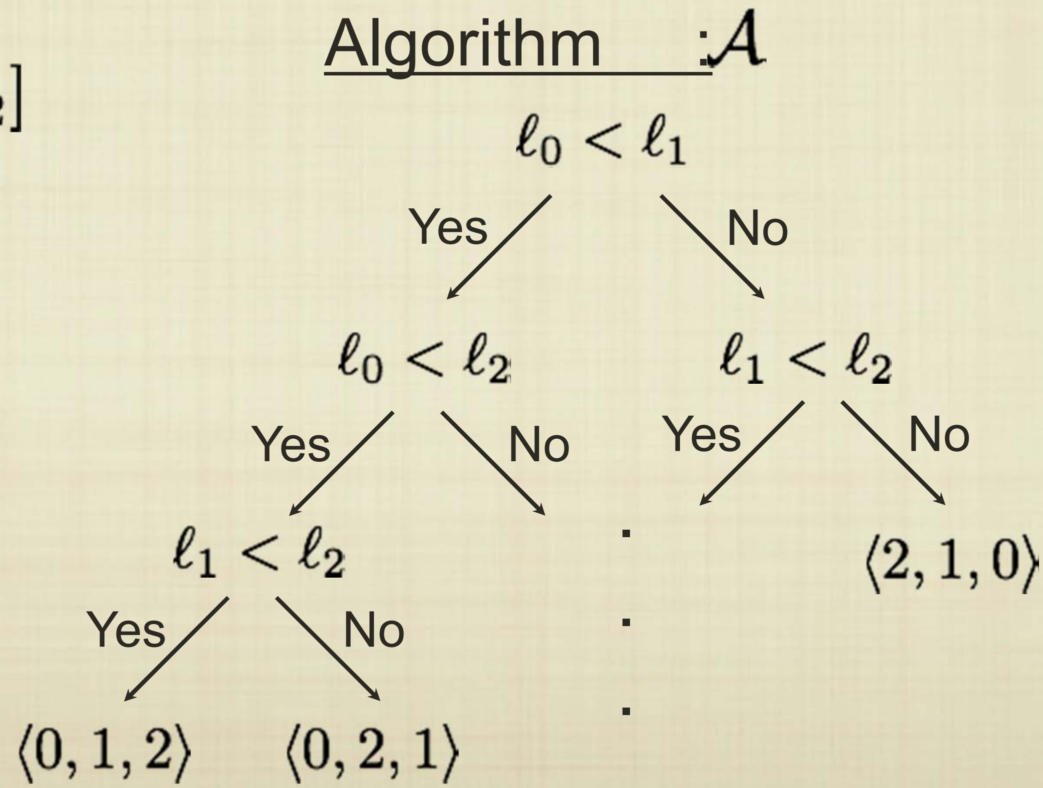
Any correct sorting algorithm must be able to permute any input into a uniquely sorted list. Therefore any sorting algorithm must be able to "apply" any of the $n!$ possible permutations necessary to produce the right answer.

Lower Bound for Sorting

Any sorting algorithm must be able to “apply” any of the $n!$ possible permutations necessary to produce the right answer.

We can visualize the behavior of any sorting algorithm as a sequence of decisions based on comparing pairs of items:

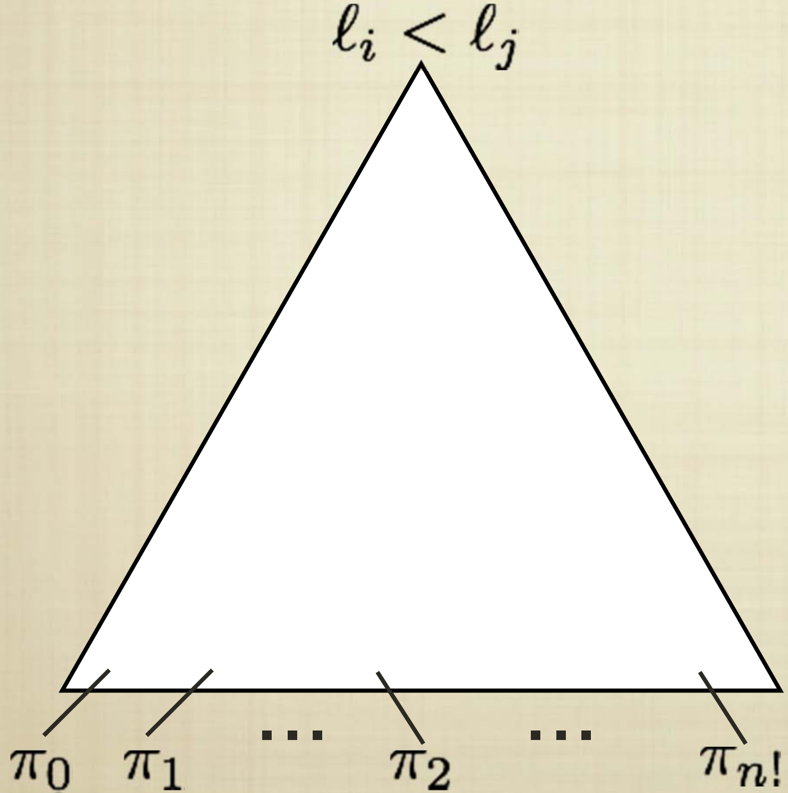
$$L = [\ell_0, \ell_1, \ell_2]$$



Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:

Algorithm: $\mathcal{A}$



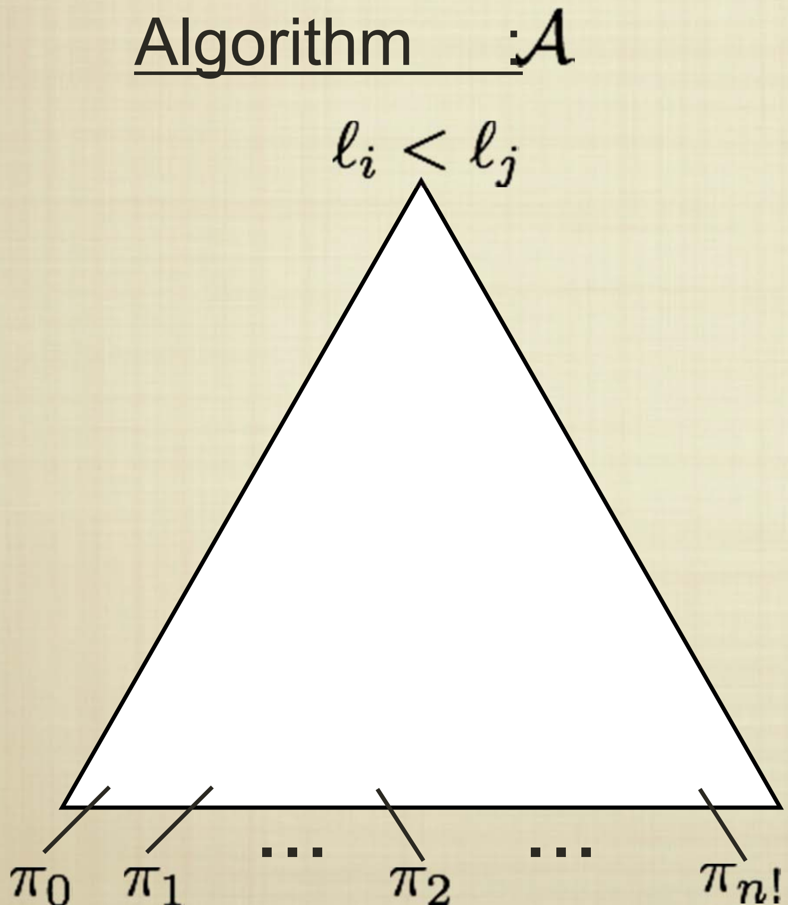
What does any of this tell us about the running time?

This decision tree is a binary tree, and its height is a lower bound on the running time of $\mathcal{A}$.

What is the minimum height of any binary decision tree?

Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:



$n! \leq$ # leaves $\leq 2^{\text{height}}$

So, $n! \leq 2^{\text{height}}$

This is equivalent to:

$\log n! \leq$ height

Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to “reach” all of these permutations by making a

The corresponding decision tree is:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

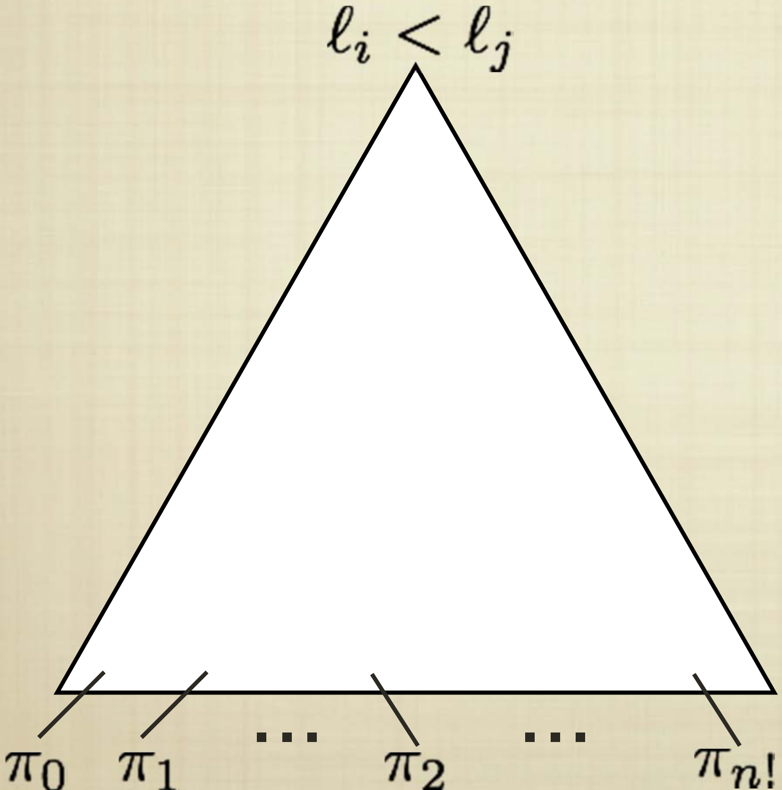
$= n \cdot \ldots \cdot (n/2+1) \cdot n/2 \cdot (n/2-1) \cdot \ldots \cdot 1$

$\geq n/2 \cdot \ldots \cdot n/2 \cdot n/2 \cdot 1 \cdot \ldots \cdot 1$

$\geq (n/2)^{n/2}$

So, $n! \geq (n/2)^{n/2}$

<u>Algorithm</u> : $\mathcal{A}$



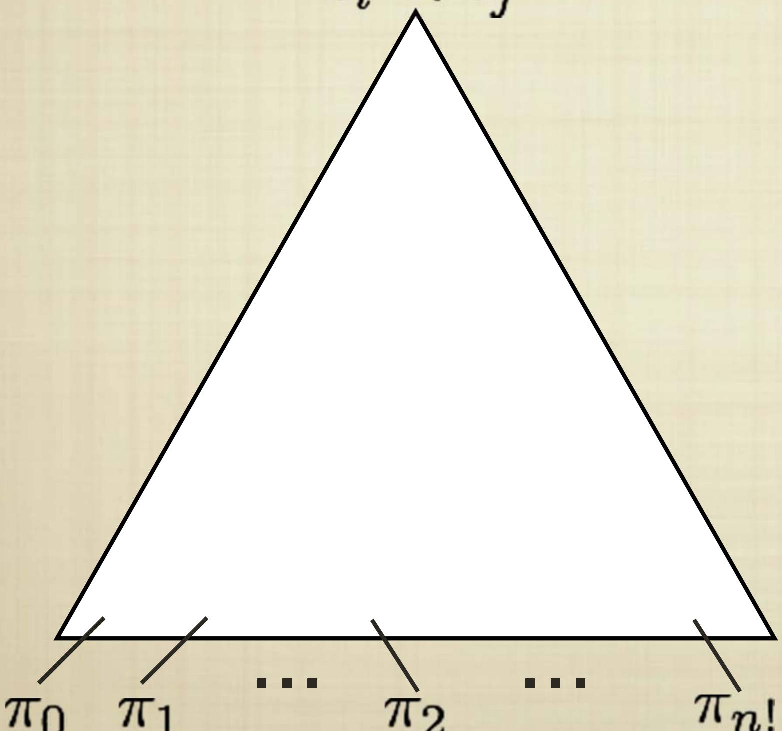
Lower Bound for Sorting

So, $n! \geq (n/2)^{n/2}$

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:

<u>Algorithm</u> : $\mathcal{A}$

$\ell_i < \ell_j$



$n! \leq$ # leaves $\leq 2^{\text{height}}$

So, $n! \leq 2^{\text{height}}$

This is equivalent to:

$\log n! \leq$ height

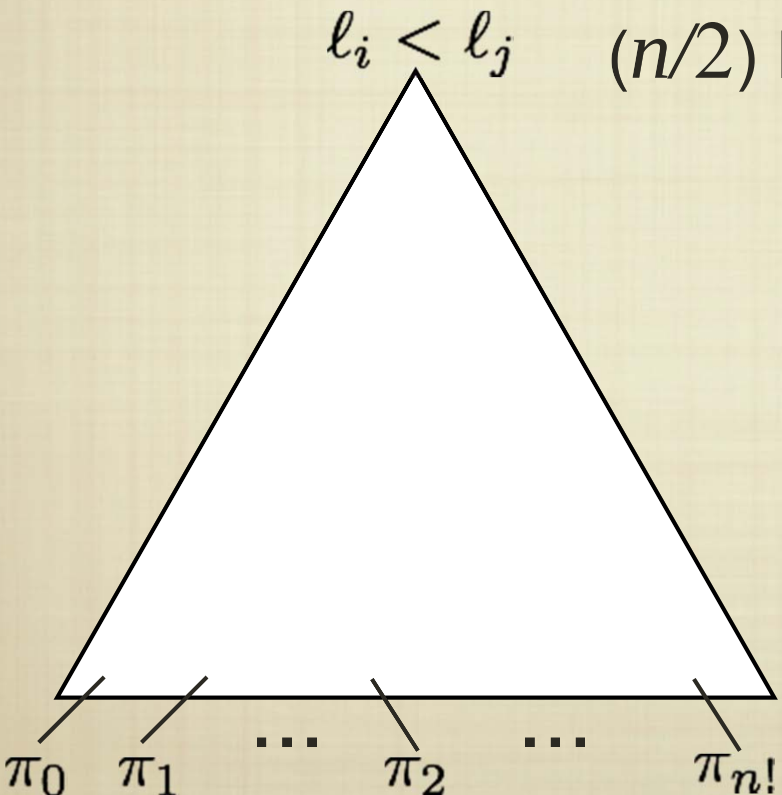
So:

$\log (n/2)^{n/2} \leq \log n! \leq$ height

Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:

Algorithm $:\mathcal{A}$



So:

$$(n/2)\log(n/2) = \log(n/2)^{n/2} \leq \log n! \leq \text{height}$$

Therefore:

$$(n/2)\log(n/2) \leq \text{height}$$

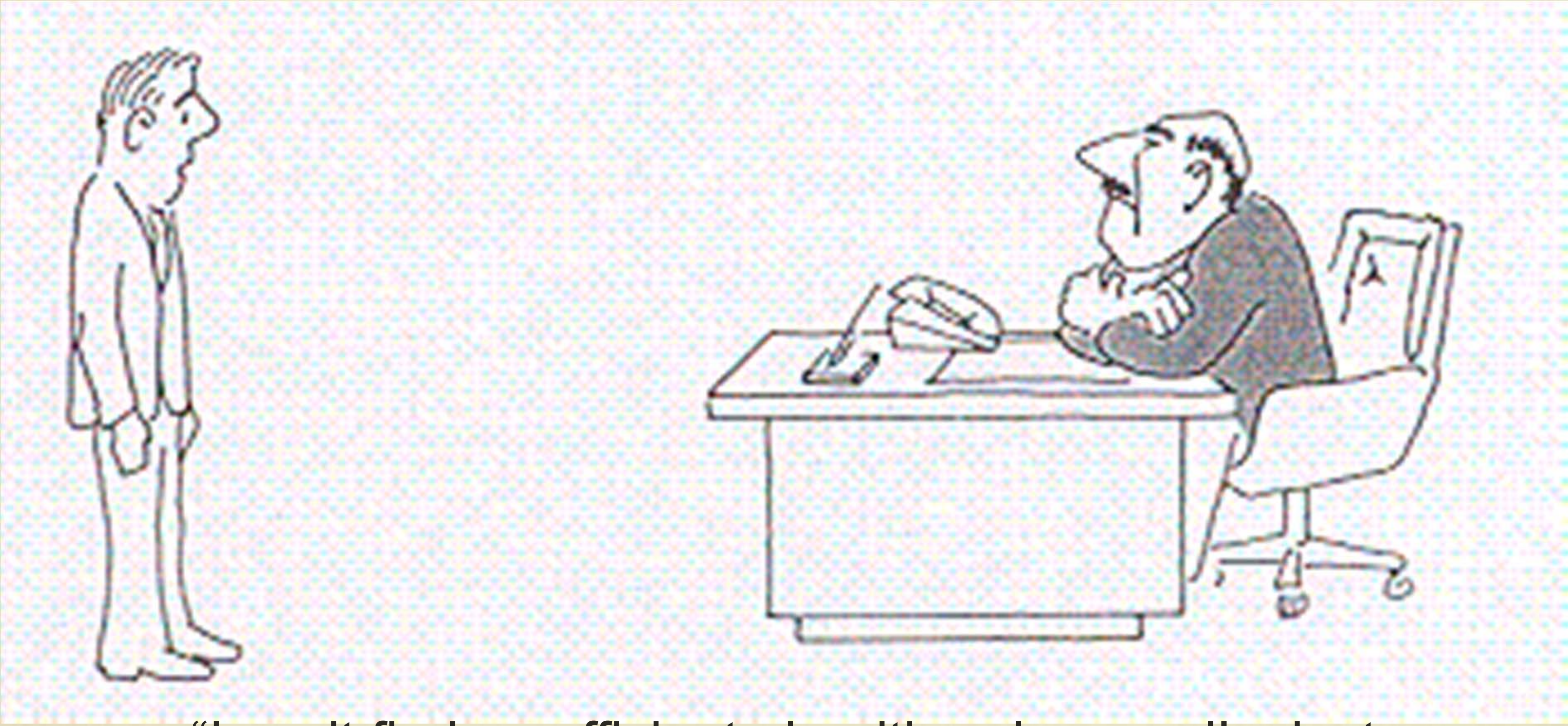
Or equivalently:

$$(1/2)\, n \log n - (n/2) \leq \text{height}$$

What does this tell us about Merge Sort?

The Power of Lower Bounds

Exponential-time Algorithm, Trivial lower bound



"I can't find an efficient algorithm, I guess I'm just dumb."

[Garey and Johnson '79]

The Power of Lower Bounds

Matching Exponential-time bounds

"I can't find an efficient algorithm, because no such algorithm is possible."

[Garey and Johnson '79]